A Methodology for the Measurement of Test Effectiveness

John C. Munson
Computer Science Department
University of Idaho
Moscow, ID 83844-1010
jmunson @cs.uidaho.edu

Allen P. Nikora
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109-8099
Allen.P.Nikora@jpl.nasa.gov

Abstract

In developing a software system, we would like to estimate the total number of faults inserted into that system, its residual fault content at any given time, and the efficacy of the testing activity in executing the code containing the newly inserted faults. Prior to test, however, there may be very little direct information regarding the number and location of faults. This lack of direct information requires the development of a fault surrogate from which the number of faults and their location can be estimated. We develop a fault surrogate based on changes in relative complexity, a synthetic measure which has been successfully used as a fault surrogate in previous work. We show that changes in the relative complexity can be used to estimate the rates at which faults are inserted into a system between successive revisions. These rates can be used to continuously monitor the total number of faults inserted into a system. Finally, we develop a method for determining test effectiveness based on measuring the proportion of testing activity devoted to exercising those areas of the system that have changed since the last version.

1 Introduction

Over a number of years of study, we can now establish a distinct relationship between software faults and certain aspects of software complexity. When a software system consisting of many distinct software modules is built for the first time, we have little or no direct information as to the location of faults in the code. Some of the mod-

ules will have far more faults in them than others. We now, however, know that the number of faults in a module is highly correlated with certain software attributes that can be measured. This means that we can measure the software on these specific attributes and have some reasonable notion as to the degree to which the modules are fault prone [9,13].

In the absence of information as to the specific location of software faults, we have successfully used a derived metric, the relative complexity measure, as a fault surrogate. That is, if the relative complexity value of a module is large, then it will likely have a large number of latent faults. If, on the other hand, the relative complexity of a module is small, then it will tend to have fewer faults. As the software system evolves through a number of sequential builds, faults will be identified and the code will be changed in an attempt to eliminate the identified faults. The introduction of new code however, is a fault prone process just as the initial code generation was. Faults may well be injected during this evolutionary process.

Code does not always change just to fix faults that have been isolated in it. Some changes to code during its evolution represent enhancements, design modifications, or changes in the code in response to continually evolving requirements. These incremental code enhancements may also result in the insertion of still more faults. Thus, as a system progresses through a series of builds, the relative complexity fault surrogate of each program module that has been altered must also change. We will see that the rate of change in relative complexity will serve as a good index of the rate of fault insertion.

Once the rate of fault insertion has been established, it becomes possible to estimate the number of faults remaining in the system at any point during the development. Since we use changes in relative complexity as an index of the fault insertion rate, it becomes possible to estimate the number of residual faults at the module level, in which a module is taken to be a procedure, function, or method. This information is useful to software development managers wishing to estimate the resources required to remove the remaining faults – not only can the number of remaining faults be estimated, but it is possible to direct fault detection and removal resources at those portions of the software estimated to have the highest concentrations of residual faults. However, this is only half of the picture. Once the software is operating in

the field, we wish to estimate its reliability. The estimated number of residual faults, a static measure, must be transformed into an estimate of the system's dynamic behavior.

The general notion of software test is that rate of fault removal will generally exceed the rate of fault insertion. In most cases, this is probably true [15]. Some changes are rather more heroic than others. During these more substantive change cycles, it is quite possible that the actual number of faults in the system will rise. We would be very mistaken, then, to assume that software test will monotonically reduce the number of faults in a system. This will only be the case when the rate of fault removal exceeds the rate of fault insertion. The rate of fault removal is relatively easy to measure. The rate of fault insertion is much more tenuous. This fault insertion process is directly related to two measures that we can take on code as it evolves, code change and code churn.

In this investigation we will establish a methodology whereby code can be measured from one build to the next, a measurement baseline. We will use this measurement baseline to develop an assessment of the rate of change to a system as measured by our relative complexity fault surrogate. From this change process we will then be able to derive a direct measure of the rate of fault insertion based on changes in the software from one build to the next. We examine data from an actual system on which faults may be traced to specific build increments to assess the predicted rate of fault insertion with the actual. Finally, we will develop a method of measuring the efficiency of a test activity.

To estimate rates of fault insertion, it is necessary to identify a complete software system on which every version of every module has been archived together with the faults that have been recorded against the system as it evolved. Of the two systems we analyzed for this study, the Cassini Orbiter Command and Data Subsystem at JPL met all of our objectives. On the first build of this system there were approximately 96K source lines of code in approximately 750 program modules. On the last build there were approximately 110K lines of source code in approximately 800 program modules. As the system progressed from the first to the last build there were a total of 45,200 different versions of these modules. On the average, then, each module progressed through an average of 56 evolutionary steps or versions. For the purposes of this study, the Ada program module is a procedure

or function. It is the smallest unit of the Ada language structure that may be measured. A number of modules present in the first build of the system were removed on subsequent builds. Similarly, a number of modules were added.

The Cassini CDS does not represent an extraordinary software system. It is quite typical of the amount of change activity that will occur in the development of a system on the order of 100 KLOC. It is a non-trivial measurement problem to track the system as it evolves. Again, there are two different sets of measurement activities that must occur at once. We are interested in the changes in the source code and we are interested in the fault reports that are being filed against each module.

To determine the efficiency of a test activity, it is necessary to have a system in which structural changes between one increment and its predecessor can be measured together with the execution profile observed during test. Since we were unable to accomplish this for the CASSINI CDS flight software, we studied the real-time software for a commercial embedded system.

2 A Measurement Baseline

The measurement of an evolving software system through the shifting sands of time is not an easy task. Perhaps one of the most difficult issues relates to the establishment of a baseline against which the evolving systems may be compared. This problem is very similar to that encountered by the surveying profession. If we were to buy a piece of property, there are certain physical attributes that we would like to know about. Among these properties is the topology of the site. To establish the topological characteristics of the land, we will have to seek out a benchmark. This benchmark represents an arbitrary point somewhere on the subject property. The distance and the elevation of every other point on the property may then be established in relation to the measurement baseline. Interestingly enough, we can pick any point on the property, establish a new baseline, and get exactly the same topology for the property. The property does not change. Only our perspective changes.

When measuring software evolution, we need to establish a measurement baseline for the same purpose described above [17,14].

We need a fixed point against which all others can be compared. Our measurement baseline also needs to maintain the property so that, when another point is chosen, the exact same picture of software evolution emerges; only the perspective changes. The individual points involved in measuring software evolution are individual builds of the system.

Standardizing metrics for one particular build is simple. For each metric obtained for each module, subtract from that metric its mean and divide by its standard deviation. This puts all of the metrics on the same relative scale, with a mean of zero and a standard deviation of one. This works fine for comparing modules within one particular build. But when we standardize subsequent builds using the means and standard deviations for those builds a problem arises. The standardization masks the change that has occurred between builds. In order to place all the metrics on the same relative scale and to keep from losing the effect of changes between builds, all build data is standardized using the means and standard deviations for the metrics obtained from the baseline system. This preserves trends in the data and lets measurements from different builds be compared.

For each raw metric in the baseline build, we may compute a mean and a standard deviation. Let us denote the vector of mean values for the baseline build as $\bar{\mathbf{x}}^B$ and the vector of standard deviations as $\mathbf{s}^B$. The standardized baseline metric values for any module *j* in an arbitrary build *i*, then, may be derived from raw metric values as

$$z_j^{B,i} = \frac{w_j^{B,i} - \bar{x}^{B,i}}{s^{B,i}}$$

The process of standardizing the raw metrics certainly makes them more tractable. Among other things, it now permits the comparison of metric values from one build to the next. This standardization does not solve the main problem. There are too many metrics collected on each module over many builds. We need to reduce the dimensionality of the problem. We have successfully used principal components analysis for reducing the dimensionality of the problem [10,7]. The principal components technique will reduce a set of highly correlated metrics to a much smaller set of uncorrelated or orthogonal measures. One of the products of the principal components technique is an orthogonal transformation matrix **T** that will send the standardized scores (the matrix **z**) onto a reduced set of domain scores thusly, $\mathbf{d} = \mathbf{zT}$.

In the same manner as the baseline means and standard deviations were used to transform the raw metric of any build relative to a baseline build, the transformation matrix $\mathbf{T}^B$ derived from the baseline build will be used in subsequent builds to transform standardized metric values obtained from that build to the reduced set of domain metrics as follows: $\mathbf{d}^{B,i} = \mathbf{z}^{B,i}\,\mathbf{T}^B$, where $\mathbf{z}^{B,i}$ are the standardized metric values from build i baselined on build B.

Another artifact of the principal components analysis is the set of eigenvalues that are generated for each of the new principal components. Associated with each of the new measurement domains is an eigenvalue, λ. These eigenvalues are large or small varying directly with the proportion of variance explained by each principal component. We have successfully exploited these eigenvalues to create a new metric called relative complexity, ρ, that is the weighted sum of the domain metrics to wit:

$$\rho_i = 50 + 10\sum_{j=1}^{m} \lambda_j d_j ,$$

where m is the dimensionality of the reduced metric set [10].

As was the case for the standardized metrics and the domain metrics, relative complexity may be baselined as well using the eigenvalues and the baselined domain values as follows:

$$\rho_i^B = \sum_{j=1}^{m} \lambda_j^B d_j^B$$

If the raw metrics that are used to construct the relative complexity metric are carefully chosen for their relationship to software faults, then the relative complexity metric will vary in exactly the same manner as the faults [12]. The relative complexity metric in this context is a fault surrogate. Whereas we cannot measure the faults in a program directly we can measure the relative complexity of the program modules that contain the faults. Those modules having a large relative complexity value will ultimately be found to be those with the largest number of faults [11].

3 Software Evolution

As a software system grows and modifications are made, the modules that comprise the system are recompiled and a new version, or build, is created. Each build is constructed from a distinct set of these software modules, though not always exactly the same ones. The new version may contain some of the same modules as the previous version. Some entirely new modules may even omit some modules that were present in an earlier version. Of the modules that are common to both the old and new version, some may have undergone modification since the last build. The set of modules that constitute the system on any one build is subject to material change over the life of the system.

3.1 Module Sets and Versions

When evaluating the change that occurs to the system between any two builds *i*, and *j*, we are interested in three sets of modules. The first set, $M_c^{i,j}$, is the set of modules present in both builds of the system. These modules may have changed since the earlier version but were not removed. The second set, $M_a^{i,j}$, is the set of modules that were in the early build and were removed prior to the later build. The final set, $M_b^{i,j}$, is the set of modules that have been added to the system since the earlier build.

As an example, let build *i* consist of the following set of modules.

$$M^i = \{m_1, m_2, m_3, m_4, m_5\}$$

Between build *i* and *j* module m_3 is removed giving

$$\begin{aligned} M^j &= M^i \cup M_b^{i,j} - M_a^{i,j} \\ &= \{m_1, m_2, m_3, m_4, m_5\} \cup \{\} - \{m_3\} \\ &= \{m_1, m_2, m_4, m_5\} \end{aligned}$$

Then between builds *j* and *k* two new modules, m_7 and m_8 are added and module m_2 is deleted giving

$$\begin{aligned} M^k &= M^j \cup M_b^{j,k} - M_a^{j,k} \\ &= \{m_1, m_2, m_4, m_5\} \cup \{m_7, m_8\} - \{m_2\} \\ &= \{m_1, m_4, m_5, m_7, m_8\} \end{aligned}$$

With a suitable baseline in place, and the module sets defined above, it is now possible to measure software evolution across a full spectrum of software metrics. We can do this first by comparing aver-

age metric values for the different builds. Secondly, we can measure the increase or decrease in system complexity as measured by the code delta, or we can measure the total amount of change the system has undergone between builds, code churn.

We can now see that establishing the complexity of a system across builds in the face of changing modules and changing sets of modules is in itself a very complex problem. In terms of the example above, the relative complexity of the system $R^{B,i}$ at build i, the early build, is given by

$$R^{B,i} = \sum_{m_c \in M^i} \rho_c^{B,i},$$

where $\rho_c^{B,i}$ is the relative complexity of module m_c on this build baselined by build B.

Similarly, the relative complexity of the system $R^{B,j}$ at build j, the latter build is given by

$$R^{B,j} = \sum_{m_c \in M^j} \rho_c^{B,j}.$$

The later system build is said to be more complex if $R^{B,j} > R^{B,i}$. Regardless of which metric is chosen, the goal is the same. We wish to assess how the system has changed over time with respect to that particular measurement. The concept of a code delta provides this information. A code delta is, as the name implies, the difference between two builds as to the relative complexity metric.

3.2 Code Churn and Code Deltas

The change in the relative complexity in a single module between two builds may be measured in one of two distinct ways. First, we may simply compute the difference in the module relative complexity between build i and build j. We will call this value the code delta for the module m_a, or $\delta_a^{i,j} = \rho_a^{B,j} - \rho_a^{B,i}$. The absolute value of the code delta is a measure of code churn. In the case of code churn, what is important is the absolute measure of the nature that code has been modified. From the standpoint of fault insertion, removing a lot of code is probably as catastrophic as adding a bunch. The new meas-

ure of code churn, χ, for module m_a is simply $\chi_a^{i,j} = \left|\delta_a^{i,j}\right| = \left|\rho_a^{B,j} - \rho_a^{B,i}\right|$.

It is now possible to compute the total change activity for the aggregate system across all of the program modules. The total net change of the system is the sum of the code delta's for a system between two builds i and j is given by

$$\Delta^{i,j} = \sum_{m_c \in M^i} \delta_c^{i,j} - \sum_{m_a \in M_a^{i,j}} \rho_a^{B,i} + \sum_{m_b \in M_b^{i,j}} \rho_b^{B,j} .$$

With a suitable baseline in place, and the module sets defined above, it is now possible to measure software evolution across a full spectrum of software metrics. We can do this first by comparing average metric values for the different builds. Secondly, we can measure the increase or decrease in system complexity as measured by a selected metric, code delta, or we can measure the total amount of change the system has undergone between builds, code churn.

A limitation of measuring code deltas is that it doesn't give an indicator as to how much change the system has undergone. If, between builds, several software modules are removed and are replaced by modules of roughly equivalent complexity, the code delta for the system will be close to zero. The overall complexity of the system, based on the metric used to compute deltas, will not have changed much. However, the reliability of the system could have been severely affected by the process of replacing old modules with new ones. What we need is a measure to accompany code delta that indicates how much change has occurred. Code churn is a measurement, calculated in a similar manner to code delta, that provides this information. The net code churn of the same system over the same builds is

$$\nabla^{i,j} = \sum_{m_c \in M_c} \chi_c^{i,j} + \sum_{m_a \in M_a^{i,j}} \rho_a^{B,i} + \sum_{m_b \in M_b^{i,j}} \rho_b^{B,j} .$$

When several modules are replaced between builds by modules of roughly the same complexity, code delta will be approximately zero but code churn will be equal to the sum of the value of ρ for all of the modules, both inserted and deleted. Both the code delta and code churn for a particular metric are needed to assess the evolution of a system.

4 Obtaining Average Build Values

One synthetic software measure, relative complexity, has clearly been established as a successful surrogate measure of software faults [10]. It seems only reasonable that we should use it as the measure against which we compare different builds. Since relative complexity is a composite measure based on the raw measurements, it incorporates the information represented by *LOC*, *V(g)*, η_1, η_2, and all the other raw metrics of interest. Relative complexity is a single value that is representative of the complexity of the system which incorporates all of the complexity attributes we have measured (e.g., size, control flow, style, data structures, etc.).

By definition, the average relative complexity, $\bar{\rho}$, of the baseline *system* will be

$$\bar{\rho}^B = \frac{1}{N^B} \sum_{i=1}^{N^B} \rho_i^B = 50 ,$$

where N^B is the cardinality of the set of modules on build *B*, the baseline build. Relative complexity for the baseline build is calculated from standardized values using the mean and standard deviation from the baseline metrics. The relative complexities are then scaled to have a mean of 50 and a standard deviation of 10. For that reason, the average relative complexity for the baseline system will always be a fixed point. Subsequent builds are standardized using the means and standard deviations of the metrics gathered from the baseline system to allow comparisons. The average relative complexity for subsequent builds is given by

$$\bar{\rho}^k = \frac{1}{N^k} \sum_{i=1}^{N^k} \rho_i^{B,k} ,$$

where N^k is the cardinality of the set of program modules in the k^{th} build and $\rho_i^{B,k}$ is the baselined relative complexity for the i^{th} module of that set.

The total relative complexity, R^0, of a system on its initial build is simply the sum of all relative complexities of each module of the initial system,

$$R^0 = \sum_{i=1}^{N} \rho_i^0 .$$

The principle behind relative complexity is that it serves as a fault surrogate. That is, it will vary in precisely the same manner as do software faults. The fault potential r_i^0 of a particular module i is directly proportional to its value of the relative complexity fault surrogate. Thus,

$$r_i^0 = \rho_i^0 / R^0 .$$

To derive a preliminary estimate for the actual number of faults per module we may make judicious use of historical data. From previous software development projects it is possible to develop a proportionality constant, say k, that will allow the total system relative complexity to map to a specific system fault count as follows: $F^0 = kR^0$ or $R^0 = k / F^0$. Substituting for R in the previous equation, we find that

$$r_i^0 = k\rho_i^0 / F^0 .$$

Thus, our best estimate for the number of faults in module i in the initial configuration of the system is

$$g_i^0 = r_i^0 F^0 .$$

After an interval of testing a number of faults will be found and fixes made to the code to remedy the faults. Let F^j be the total number of faults found in the total system up to and including the j^{th} build of the software. In a particular module i there will be f_i^1 faults found in the first build that are attributable to this module. The estimated number of faults remaining in module i will then be

$$g_i^1 = g_i^0 - f_i^1 ,$$

assuming that we have only fixed faults in the code and not added any new ones.

Our ability to locate the remaining faults in a system will relate directly to our exposure to these faults. If, for example, at the j^{th} build of a system there are g_i^j remaining faults in module i, we can not expect to identify any of these faults unless some test activity is allocated to exercising module i.

As the code is modified over time, faults will be found and fixed. However, new faults will be introduced into the code as a result of the change. In fact, this fault insertion process is directly propor-

tional to change in the program modules from one version to the next. As a module is changed from one build to the next in response to evolving requirements changes and fault reports, its complexity will also change. Generally, the net effect of a change is that complexity will increase. Only rarely will its complexity decrease. It is now necessary to describe the measurement process for the rate of change in an evolving system.

5 Software Evolution and the Fault Insertion Process

Initially, our best estimate for the number of faults in module i in the initial configuration of the system is

$$g_i^0 = r_i^0 F^0 .$$

As the i^{th} module was tested during the test activity of the first build, the number of faults found and fixed in this process was denoted by f_i^1. However, in the process of fixing this fault, the source code will change. In all likelihood, so, too, will the relative complexity of this module. Over a sequence of builds, the complexity of this module may change substantially. Let,

$$\Delta_i^{0,j} = \sum_{k=1}^{j} \Delta_i^{k-1,k}$$

represent the net change in relative complexity to the i^{th} module over the first j builds. Then the cumulative churn in the total system over these j builds will be,

$$\nabla^{0,j} = \sum_{i=1}^{N_j} \nabla_i^{0,j} ,$$

where N_j is the cardinality of the set of all modules that were in existence over these j builds. The complexity of the i^{th} module will have changed over this sequence of builds. Its new value will be $\rho_i + \Delta_i^{0,j}$. Some changes may increase the relative complexity of this module and others may decrease it. A much better (as will be demonstrated) measure of the cumulative change to the system will be $\rho_i + \nabla_i^{0,j}$. The system complexity, R, will also have changed. Its new value will be $R^0 + \Delta^{0,j}$.

On the initial build of the system the initial burden of faults in a module was proportional to the relative complexity of the module. As the build cycle continues, the rate of fault insertion is most closely associated with the code churn. Thus, the proportion of faults in the i^{th} module will have changed over the sequence of j builds, related to its initial relative complexity and its subsequent code churn. Its new value will be

$$r_i^j = (\rho_i^0 + \nabla_i^{0,j}) \Big/ (R^0 + \nabla^{0,j})$$

We now observe that our estimate of the number of faults in the system has now changed. On the j^{th} build there will no longer be F^0 faults in the system. New faults will have been introduced as the code has evolved. In all likelihood, the initial software development process and subsequent evolution processes will be materially different. This means that there will be a different proportionality constant, say k', representing the rate of fault insertion for the evolving system. For the total system, then, there will have been $F^j = kR^0 + k'\Delta^{0,j}$ faults introduced into the system from the initial build through the j^{th} build. Each module will have had $h_i^j = r_i^j F^j$ faults introduced in it either from the initial build or on subsequent builds. Thus, our revised estimate of the number of faults remaining in module i on build j will be

$$g_i^j = h_i^j - f_i^j .$$

The rate of fault insertion is directly related to the change activity that a module will receive from one build to the next. At the system level, we can see that the expected number of injected faults from build j to build $j+1$ will be

$$\begin{aligned} F^{j+1} - F^j &= kR^0 + k'\nabla^{0,j+1} - kR^0 + k'\nabla^{0,j} \\ &= k'(\nabla^{0,j+1} - \nabla^{0,j}) \\ &= k'\nabla^{j,j+1} \end{aligned} .$$

At the module level, the rate of fault insertion will again be proportional to the level of change activity. Hence, the expected number of injected faults between build j to build $j+1$ on module i will be simply $h_i^{j+1} - h_i^j$.

The two proportionality constants k and k' are the ultimate criterion measures of the software development process and software maintenance processes. Each process has an associated fault insertion

proportionality constant. If we institute a new software development process and observe a significant change downward in the constant k, then the change would have been a good one. Very frequently, however, software processes are changed because development fads change and not because a criterion measure has indicated that a new process is superior to a previous one. We will consider that an advance in software development process has occurred if either k or k' has diminished for that new process.

6 Definition of a Fault

Unfortunately there is no particular definition of just precisely what a software fault is. In the face of this difficulty it is rather hard to develop meaningful associative models between faults and metrics. In calibrating our model, we would like to know how to count faults in an accurate and repeatable manner. In measuring the evolution of the system to talk about rates of fault introduction and removal, we measure in units to the way that the system changes over time. Changes to the system are visible at the module level, and we attempt to measure at that level of granularity. Since the measurements of system structure are collected at the module level (by module we mean procedures and functions), we would like information about faults at the same granularity. We would also like to know if there are quantities that are related to fault counts that can be used to make our calibration task easier.

Following the second definition of fault in [3,4], we consider a fault to be a **structural imperfection** in a software system that **may** lead to the system's eventually failing. In other words, it is a **physical characteristic** of the system of which the type and extent may be measured using the same ideas used to measure the properties of more traditional physical systems. Faults are introduced into a system by people making errors in their tasks - these errors may be errors of commission or errors of omission.

In order to count faults, we needed to develop a method of identification that is repeatable, consistent, and identifies faults at the same level of granularity as our structural measurements. In analyzing the flight software for the CASSINI project the fault data and the source code change data were available from two different systems.

The problem reporting information was obtained from the JPL institutional problem reporting system. For the software used in this study, failures were recorded in this system starting at subsystem-level integration, and continuing through spacecraft integration and test. Failure reports typically contain descriptions of the failure at varying levels of detail, as well as descriptions of what was done to correct the fault(s) that caused the failure. Detailed information regarding the underlying faults (e.g., where were the code changes made in each affected module) is generally unavailable from the problem reporting system.

The entire source code evolution could be obtained directly from the Software Configuration Control System (SCCS) files for all versions of the flight software. The way in which SCCS was used in this development effort makes it possible to track changes to the system at a module level in that each SCCS file stores the baseline version of that file (which may contain one or more modules) as well as the changes required to produce each subsequent increment (SCCS delta) of that file. When a module was created, or changed in response to a failure report or engineering change request, the file in which the module is contained was checked into SCCS as a new delta. This allowed us to track changes to the system at the module level as it evolved over time. For approximately 10% of the failure reports, we were able to identify the source file increment in which the fault(s) associated with a particular failure report were repaired. This information was available either in the comments inserted by the developer into the SCCS file as part of the check-in process, or as part of the set of comments at the beginning of a module that track its development history.

Using the information described above, we performed the following steps to identify faults:

- For each problem report, search all of the SCCS files to identify all modules and the increment(s) of each module for which the software was changed in response to the problem report.
- For each increment of each module identified in Step 1, start with the assumption that all differences between the increment in which repairs are implemented and the previous increment are due solely to fault repair. Note that this is not necessarily a valid assumption - developers may be making functional enhancements to the system in the same increment that fault repairs are being made. Careful analysis of failure reports for

which there was sufficiently detailed descriptive information served to separate areas of fault repair from other changes. However, the level of detail required to perform this analysis was not consistently available.

- Use a differential comparator (e.g.: Unix `diff`) to obtain the differences between the increment(s) in which the fault(s) were repaired, and the immediately preceding increment(s). The results indicated the areas to be searched for faults.

After completing the last step, we still had to identify and count the faults - the results of the differential comparison cannot simply be counted up to give a total number of faults. In order to do this, we developed a taxonomy for identifying and counting faults [Niko98]. This taxonomy differs from others in that it does not seek to identify the root cause of the fault. Rather, it is based on the types of changes made to the software to repair the faults associated with failure reports - in other words, it constitutes an operational definition of a fault. Although identifying the root causes of faults is important in improving the development process [1, 5], it is first necessary to identify the faults. We do not claim that this is the only way to identify and count faults, nor do we claim that this taxonomy is complete. However, we found that this taxonomy allowed us to successfully identify faults in the software used in the study in a consistent manner at the appropriate level of granularity.

7 The Relationship Between Faults And Code Changes

Having established a theoretical relationship between software faults and code changes, it is now of interest to validate this model empirically. This measurement occurred on two simultaneous fronts. First, all of the versions of all of the source code modules were measured. From these measurements, code churn and code deltas were obtained for every version of every module. The failure reports were sampled to lead to specific faults in the code. These faults were classified according to the above taxonomy manually on a case by case basis. Then we were able to build a regression model relating the code measures to the code faults.

The Ada source code modules for all versions of each of these modules were systematically reconstructed from the SCCS code deltas.

Each of these module versions was then measured by the UX-Metric analysis tool for Ada [19]. Not all metrics provided by this tool were used in this study. Only a subset of these actually provide distinct sources of variation [6]. The specific metrics used in this study are shown in Table 1.

Table 1. Software Metric Definitions

Metrics	Definition
η_1	Count of unique operators [2]
η_2	Count of unique operands
N_1	Count of total operators
N_2	Count of total operands
P/R	Purity ratio: ratio of Halstead's $\hat{N}$ to total program vocabulary
V(g)	McCabe's cyclomatic complexity
Depth	Maximum nesting level of program blocks
AveDepth	Average nesting level of program blocks
LOC	Number of lines of code
Blk	Number of blank lines
Cmt	Count of comments
CmtWds	Total words used in all comments
Stmts	Count of executable statements
LSS	Number of logical source statements
PSS	Number of physical source statements
NonEx	Number of non-executable statements
AveSpan	Average number of lines of code between references to each variable
Vl	Average variable name length

To establish a baseline system, all of the metric data for the module versions that were members of the first build of CDS were then

analyzed by our PCA-RCM tool. This tool is designed to compute relative complexity values either from a baseline system or from a system being compared to the baseline system. In that the first build of the Cassini CDS system was selected to be the baseline system, the PCA-RCM tool performed a principal components analysis on these data with an orthogonal varimax rotation. The objective of this phase of the analysis is to use the principal components technique to reduce the dimensionality of the metric set.

Table 2. Principal Components of Software Metrics

Metric	Size	Structure	Style	Nesting
Stmts	**0.968**	0.022	-0.079	0.021
LSS	**0.961**	0.025	-0.080	0.004
N_2	**0.926**	0.016	0.086	0.086
N_1	**0.934**	0.016	0.074	0.077
η_2	**0.884**	0.012	-0.244	0.043
AveSpan	**0.852**	0.032	0.031	-0.082
V(g)	**0.843**	0.032	-0.094	-0.114
η_1	**0.635**	-0.055	-0.522	-0.136
Depth	**0.617**	-0.022	-0.337	-0.379
LOC	-0.027	**0.979**	0.136	0.015
Cmt	-0.046	**0.970**	0.108	0.004
PSS	-0.043	**0.961**	0.149	0.019
CmtWds	0.033	**0.931**	0.058	-0.010
NonEx	-0.053	**0.928**	0.076	-0.009
Blk	0.263	**0.898**	0.048	0.005
P/R	-0.148	-0.198	**-0.878**	0.052
Vl	0.372	-0.232	**-0.752**	0.010
AveDepth	-0.000	-0.009	0.041	**-0.938**

Metric	Size	Structure	Style	Nesting
% Variance	37.956	30.315	10.454	6.009

As may been seen in Table 2, there are four principal components for the 18 metrics shown in Table 1. For convenience, we have chosen to name these principal components as **Size**, **Structure**, **Style** and **Nesting**. From the last row in Table 2 we can see that the new reduced set of orthogonal components of the original 18 metrics account for approximately 85% of the variation in the original metric set.

As is typical in the principal components analysis of metric data, the **Size** domain dominates the analysis. It alone accounts for approximately 38% of the total variation in the original metric set. Not surprisingly, this domain contains the metrics of total statement count (*Stmts*), logical source statements (LSS), the Halstead lexical metric primitives of operator and operand count, but it also contains cyclomatic complexity (*V(g)*). In that we regularly find cyclomatic complexity in this domain we are forced to conclude that it is only a simple measure of size in the same manner as statement count. The **Structure** domain contain those metrics relating to the physical structure of the program such as non-executable statements (*NonEx*) and the program block count (*Blk*). The **Style** domain contains measures of attribute that are directly under a programmer's control such as variable length (*VI*) and purity ratio (*P/R*). The **Nesting** domain consist of the single metric that is a measure of the average depth of nesting of program modules (*AveDepth*).

In order to transform the raw metrics for each module version into their corresponding relative complexity values, the means and the standard deviations must be computed. These are shown in Table 3. These values will be used to transform all raw metric values for all versions of all modules to their baselined *z* score values. The last four columns of Table 3 contain the actual transformation matrix that will map the metric *z* score values onto their orthogonal equivalents to obtain the orthogonal domain metric values used in the computation of relative complexity. Finally, the eigenvalues for the four domains are presented in the last row of this table.

Table 3, then contains all of the essential information needed to obtain baselined relative complexity values for any version of any

module relative to the baseline build. As an aside, it is not necessary that the baseline build be the initial build. As a typical system progresses through hundreds of builds in the course of its life, it is well worth reestablishing a baseline closer to the current system. In any event, these baseline data are saved by the PCA-RCM tool for use in later computation of metric values. Whenever the tool is invoked referencing the baseline data it will automatically use these data to transform the raw metric values given to it.

Table 3. Baseline Transformation Data

Metric	$\bar{x}^B$	δ^B	Domain 1	Domain 2	Domain 3	Domain 4
Stmts	11.3	7.7	0.10	-0.02	0.26	0.05
LSS	25.1	27.0	0.13	0.00	0.04	-0.09
N_2	79.5	129.0	0.13	0.02	-0.17	-0.08
N_1	68.2	115.7	0.13	0.02	-0.17	-0.09
η_2	1.3	0.5	0.00	-0.07	0.54	-0.16
AveSpan	4.7	6.1	0.12	0.01	-0.03	0.07
V(g)	1.4	1.5	0.10	-0.01	0.17	0.30
η_1	0.0	0.0	0.01	0.00	0.06	0.88
Depth	162.0	515.8	-0.01	0.17	0.07	-0.02
LOC	19.0	30.1	0.03	0.16	0.07	-0.02
Cmt	34.1	124.2	-0.01	0.17	0.09	-0.01
PSS	139.2	452.4	0.00	0.16	0.10	0.00
CmtWds	16.6	20.4	0.14	0.01	-0.07	-0.05
NonEx	17.5	23.5	0.14	0.01	-0.07	-0.04
Blk	108.8	372.1	-0.01	0.17	0.06	-0.02
P/R	7.3	22.8	-0.01	0.16	0.10	0.00
Vl	5.7	8.2	0.12	0.02	-0.11	0.06

Metric	$\bar{x}^B$	δ^B	Domain 1	Domain 2	Domain 3	Domain 4
AveDepth	9.0	4.4	0.07	-0.06	0.40	-0.11
Eigen-values			6.832	5.457	1.882	1.082

In relating the number of faults inserted in an increment to measures of a module's structural change, we had only a small number of observations with which to work. Problem reports could not be consistently traced back to source code, and there were numerous modules for which UX-Metric did not report measurements. The net result was that of the over 100 faults that were initially identified, there were only 35 observations in which a fault could be associated with a particular increment of a module, and with that increment's measures of code delta and code churn.

For each of the 35 modules for which there was viable fault data, there were three data points. First, we had the number of injected faults for that module that were the direct result of changes that had occurred on that module between the current version that contained the faults and the previous version that did not. Second, we had code delta values for each of these modules from the current to the previous version. Finally, we had code churn values derived from the code deltas.

Linear regression models were computed for code churn and code deltas with code faults as the dependent variable in both cases. Both models were built without constant terms in that we surmise that if no changes were made to a module, then no new faults could be introduced. The results of the regression between faults and code deltas were not at all surprising. The squared multiple R for this model was 0.001, about as close to zero as you can get. This result is directly attributable to the non-linearity of the data. Change comes in two flavors. Change may increase the complexity of a module. Change may decrease the complexity of a model. Faults, on the other hand, are not related to the direction of the change but to its intensity. Removing masses of code from a module is just as likely to introduce faults and adding code to it.

Table 4. Regression Analysis of Variance

Source	Sum-of-Squares	DF	Mean-Square	F-Ratio	P
Regression	331.879	1	331.879	62.996	0.00
Residual	179.121	34	10.673	5.268	

Table 5. Regression Model

Effect	Coefficient	Std Error	t	P(2-Tail)
Churn	0.576	0.073	7.937	0.000

Table 6. Regression Statistics

N	Multiple R	Squared multiple R	Standard error of estimate
35	0.806	0.649	2.296

The regression model between code churn and faults is dramatically different. The regression ANOVA for this model are shown in Table 4. Whereas code deltas do not show a linear relationship with faults, code churn certainly does. The actual regression model is given in Table 5. In Table 6 the regressions statistics have been reported. Of particular interest is the Squared Multiple R term. This has a value of 0.649. This means, roughly, that the regression model will account for more that 65% of the variation in the faults of the observed modules based on the values of code churn.

Of course, it may be the case that both the amount of change and the direction in which the change occurred affect the number of faults inserted into the system. The linear regression through the origin shown in Tables 7, 8, and 9 below illustrates this particular regression model. Tables 5 and 8 contain our estimates for the constant k relating the rate of fault insertion to the measured structural change, measured by code churn and code delta. We see that the model incorporating code delta, as well as code churn, performs significantly better than the model incorporating code churn alone, as measured by Squared Multiple R and Mean Sum of Squares.

Table 7. Regression Analysis of Variance

Source	Sum-of-Squares	DF	Mean-Square	F-Ratio	P
Regression	367.247	2	183.623	42.153	0.00
Residual	143.753	33	4.356		

Table 8. Regression Model

Effect	Coefficient	Std Error	t	P(2-Tail)
Churn	0.647	0.071	9.172	0.00
Delta	0.201	0.071	2.849	0.00

Table 9. Regression Statistics

N	Multiple R	Squared multiple R	Standard error of estimate
35	.848	.719	2.08

We evaluated the predictive ability of the regression models by performing a crossvalidation. We performed a specific type of crossvalidation, excluding one observation at a time and examining the prediction made with the remaining observations. For our set of 35 observations, 35 different predictions were made for each regression model. Tables 10 and 11 summarize the crossvalidation results for the two linear regression models through the origin, which are specified in Tables 4 - 9. For each of these models, Tables 10 and 11 show statistics for:

- Predicted squared residuals. For each observation, a regression model is formed that excludes that observation. The resulting model then uses the value of the excluded observation to predict the number of faults inserted. This prediction is then subtracted from the number of faults actually observed for the excluded observation. This residual is then squared, thereby forming the predicted squared residual.

- Ratio of predicted number of faults to observed number of faults, where predictions are made for excluded observations. For each excluded observation, a prediction is made as described above. The ratio of the prediction made using each excluded observation to the actual number of faults is then formed.

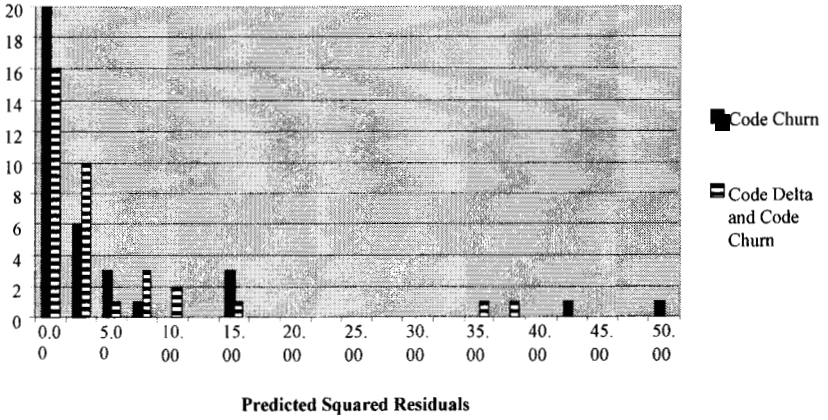
Table 10. Predicted Squared Residuals for Linear Regressions

Model	Mean	Vari-ance	Mini mum	Maxi-mum	25th %tile	50th %tile	75th %tile
$d^{j,j-1} = b_1 \nabla^{j,j}$	5.43	124.91	0.003	51.02	0.752	1.000	3.840
$d^{j,j+1} = b_1 \nabla^{j,j} + b_2 \Delta^{j,j}$	4.68	69.66	0.03	36.27	0.799	1.479	3.876

Table 11. Ratio of Predicted Faults to Observed Faults for Linear Regressions

Model	Mean	Vari-ance	Mini mum	Maxi mum	25th %tile	50th %tile	75th %tile
$d^{j,j+1} = b_1 \nabla^{j}$	0.899	1.334	0.00	5.03	8.64E-2	0.508	1.136
$d^{j,j+1} = b_1 \nabla^{j,j} + b_2 \Delta^{j,j}$	0.911	1.164	0.00	4.03	6.69E-2	0.463	1.447

Figure 1 - Histograms of Predicted Squared Residuals for Excluded Observations



Figures 1 and 2 are histograms that present additional information to that given in Tables 10 and 11. Looking at Table 10, we see that the regression model that includes both code delta and code churn has the lowest values for mean predicted squared residual and variance of the predicted squared residual. This is also shown in Figure 1. In addition, Table 10 shows that the two parameter model that includes both code churn and code delta has a slightly smaller difference between the points at the 25^{th} and 75^{th} percentiles.

Figure 2 - Histograms of Ratio of Predicted to Observed Number of Faults for Excluded Observations

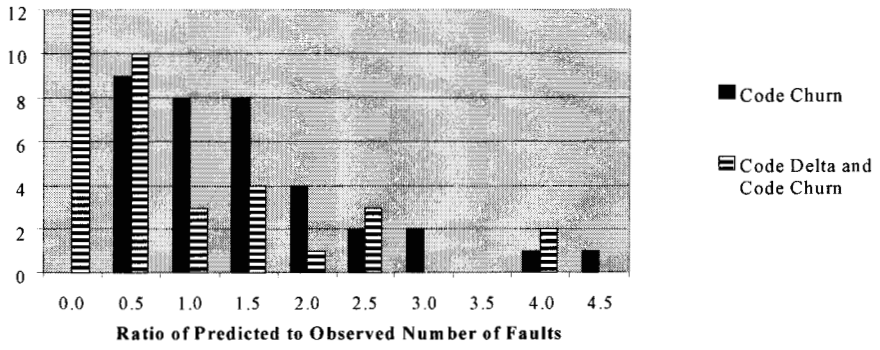


Table 11 shows that the mean value of the predictions made by the model which includes both code delta and code churn comes closer to predicting the number of faults observed. Table 11 also shows that the model which includes only code churn and code delta has the lowest variance for this ratio of predicted to actual values. This can be seen in Figure 2, which shows that the regressions depending only code churn has a higher variability for this ratio than the regression which includes both code delta and code churn. However, the range between the points at the 25^{th} and 75^{th} percentiles is the highest for the two parameter model.

Table 12 shows the results of the Wilcoxon Signed Ranks test, as applied to the predictions for the excluded observations and the number of faults observed for each of the regression models. We see that about 2/3 of the estimates tend to be less than the number of faults observed.

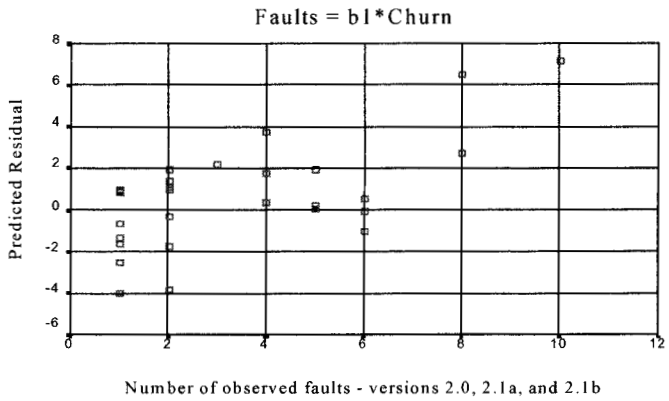
We can also plot the predicted residuals against the actual number of observed faults for each of the four linear regression models. These plots are shown in Figures 3 and 4.

Table 12. Wilcoxon Signed Ranks Test for Linear Regressions Through the Origin

Sample Pair		N	Mean Rank	Sum of Ranks	Test Statistic Z	Asymp-totic Signifi-cance (2-tailed)
Observed Faults; Churn Only	- Ranks	25[a]	17.52	438.00	-2.015[d]	.044
	+ Ranks	10[b]	19.20	192.00		
	Ties	0[c]				
	Total	35				
Observed Faults; Churn and Delta	- Ranks	24[a]	16.92	406.00	-1.491[d]	.136
	+ Ranks	11[b]	20.36	224.00		
	Ties	0[c]				
	Total	35				

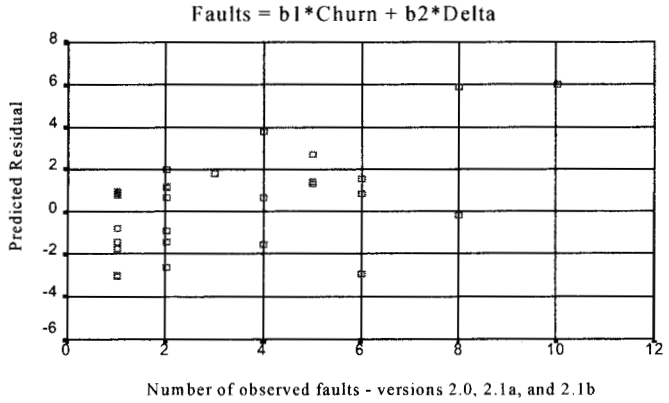
a. Observed Faults > Regression model predictions
b. Observed Faults < Regression model predictions
c. Observed Faults = Regression model predictions
d. Based on positive ranks

Figure 3 - Predicted Residuals vs. Number of Observed Faults for Linear Regression with Churn



The results of the Wilcoxon signed ranks tests, as well as Figures 3 and 4, indicate that the predictive accuracy of the regression models might be improved if syntactic analyzers capable of measuring additional aspects of a software system's structure were available.

Figure 4 - Predicted Residuals vs. Number of Observed Faults for Linear Regression with Churn and Delta



Finally, we investigated whether the linear regression model which uses code churn alone is an adequate predictor at a particular significance level when compared to the model using both code churn and code delta. We used the R^2-adequate test [8, 16] to examine the linear regression models through the origin and determine whether the model that depends only on code churn is an adequate predictor. A subset of predictor variables is said to be R^2-adequate at significance level α if:

$R^2_{sub} > 1-(1-R^2_{full})(1+d_{n,k})$, where

- R^2_{sub} is the R^2 value for the subset of predictors
- R^2_{full} is the R^2 value for the full set of predictors
- $d_{n,k} = (kF_{k,n-k-1})/n-k-1$, where
  k = number of predictor variables in the model
  n = number of observations
  F = F statistic for significance α for n,k degrees of freedom.

Table 13 shows values of R^2, k, degrees of freedom, $F_{k,n-k-1}$, $d_{n,k}$, and R^2_{sub} for both linear models through the origin. The number of observations, n, is 35, and we specify α=.05.

Table 13. Values of R^2, DOF, k, $F_{k,n-k-1}$, and $d_{n,k}$ for R^2-adequate Test

Linear Regressions Through Origin	R^2	DF	k	$F_{k,n-k-1}$ for significance α	d(n,k)	Threshold for significance α
Churn only	0.649	34	1	4.139	0.125	-----
Churn, Delta	0.719	33	2	3.295	0.206	0.661

Table 13 shows that the value of Multiple Squared R for the regression using only code churn is 0.649. The 5% significance threshold for the code churn and code delta model is 0.661. This means that the regression model using only code churn is not R^2 adequate when compared to the model using both code churn and code delta. Although the amount of change occurring between subsequent revisions appears to be the primary factor determining the number of faults inserted; the direction of that change also appears to be a significant factor.

8 Testing Objectives

Deterministically testing a large software system is virtually impossible. Trivial systems, on the order of 20 or 30 modules, often have far too many possible execution paths for complete deterministic testing. This being the case, we must revisit what we hope to accomplish by testing the system. Is our goal to remove all of the faults within the code? If this is our goal, how do we know when we have them all? What is it worth, in terms of expense, to try to find one more fault? Given unlimited time and resources, identification and removal of all faults might be a noble goal, but real world constraints make this largely unattainable. The problem is that we must provide an adequate level of reliability in light of the fact that we cannot find and remove all of the faults. Through the use of software measurement, we hope to identify which modules contain the most faults and, based on execution profiles of the system, how these potential faults can impact software reliability. The idea is that a fault that never executes, never causes a failure. However, a fault that lies along the path of normal execution will cause frequent failures. The majority of the testing effort should be spent finding those faults that are most likely to cause failure.

The first step towards this testing paradigm is the identification of those modules that are likely to contain the most faults. The objec-

tives of the software test process are not clearly specified and sometimes not clearly understood. An implicit objective of a deterministic approach to testing is to design a systematic and deterministic test procedure that will guarantee sufficient test exposure for the random faults distributed throughout a program. By insuring, for example, that all possible paths have been executed, then any potential faults on these paths will have had the opportunity to have been expressed.

We must, however, come to accept the fact that some faults will always be present in the code. We will not be able to eliminate them all. The objective of the testing process should be to find those faults that will have the greatest impact on the safety/survivability of the code. Under this view of the software testing process, the act of testing may be thought of as conducting an experiment on the behavior of the code under typical execution conditions. We will determine, a priori, exactly what we wish to learn about the code in the test process and conduct the experiment until this stopping condition has been reached.

To know the loci of probable faults in a complex software system is not a sufficient condition for reliability modeling. A software system may be viewed as a set of program modules that are executing a set of mutually exclusive functions. If the system executes a functionality expressed by a subset of modules that are fault free, it will never fail. If, on the other hand, the system is executing a functionality expressed in a subset of fault laden modules, there is a very high probability that it will fail. Thus, failure probability is dependent upon the input data sets which drive the system into regions of code (i.e., functionalities) of differing complexities (i.e., fault proneness).

Each software test suite implements a subset of functionalities. As each test is run to completion it generates a *test execution profile* which represents the results of the execution of one or more functions. When a program begins the execution of a particular functionality we can describe this beginning as the start of a stochastic process. For the system, S, there is a call tree that shows the transition of program control from one program module to another. This transition can be modeled as a stochastic process, where we define an indexed collection of random variables $\{X_t\}$, where the index t runs through a set of non-negative integers, $t = 0,1,2,\cdots$ representing the epochs of the process. At any particular epoch the software is found to be executing exactly

one of its M modules. The fact of the execution occurring in a particular module is a *state* of the system. For a given software system, it may be found in exactly one of a finite number of mutually exclusive and exhaustive states, $1,2,\cdots M$. In this representation of the system, there is a stochastic process $\{X_t\}$, where the random variables are observed at epochs $t = 0,1,2,\cdots$ and where each random variable may take on any one of the M integers, from the state space $A = 1,2,\cdots M$.

The probability that a particular module may execute is a conditional probability. Let Y be a random variable defined on the indices of the set of elements of F. Then $p_i^{(k)} = \Pr[X_T = i \mid Y = k]$ where $k = 1,2,\ldots,\#\{F\}$ represents the execution profile for a set of modules expressing function k exclusively. The distribution of the execution profile is multinomial for a software system consisting of more than two modules. In other words, for each functionality, f_i, there is an execution profile represented by the probabilities $p_1^{(i)}, p_2^{(i)}, p_3^{(i)}, \cdots, p_n^{(i)}$.

10 Test Efficiency

The test process for evolving software systems takes on a different measurement aspect than that of new systems. Existing systems are continually being modified as a normal part of the software maintenance activity. Changes will be introduced into this system based on the need for corrections, adaptations to changing requirements, and enhancements to make the system perform faster/better. The precise effects of changes to software modules in terms of number of latent faults is now reasonably well understood. From a statistical testing perspective, test effort should be focused on those modules that are most likely to contain faults. Each program module that has been modified, then, should be tested in proportion to the number of anticipated faults that might have been introduced into it.

Each program module is usually closely linked to a specific functionality. That is, as we exercise a particular functionality a distinct execution profile emerges for that functionality. For each functionality, some modules have a high probability of being executed, while others have a low probability. Each test suite will express one or more of these functionalities. The execution profiles generated from

each test may be characterized by the probability distribution $P = \{p_i \mid 1 \le i \le n\}$ for the k^{th} test.

In the face of the evolving nature of the software system, the impact of a single test may change from one build to the next. Each program module has a relative complexity value. This relative complexity is a fault surrogate. That is, the larger value of the relative complexity the greater fault potential that a module has. If a given module has a large fault potential, but limited exposure (small profile value) then the ***functional complexity*** of that module is also small. Our objective during the test phase is to maximize our exposure to the faults in the system. Another way to say this is that we wish to maximize functional complexity, ϕ , given by

$$\phi_i^{(k)} = \sum_{j=1}^{n} p_j^{(k)} \rho_j^i$$

where ρ_j^i is the relative complexity of the j^{th} module on the i^{th} system build and $p_j^{(k)}$ is the test profile of the k^{th} test suite.

The initial phase of the efficient testing of changed code is to identify the functionalities that will exercise the modules that have changed. Each of these functionalities so designated will have an associated test suite designed to exercise that functionality. With this information it is now possible to describe the efficiency of a test from a mathematical/statistical perspective. A regression test is one specifically tailored to exercise the functionalities that will cause the changed modules to be executed. A regression test will be efficient if it does a good job of exercising changed code. It is worth noting, however, that a regression test that is efficient on one build may be inefficient on a subsequent build. The efficiency of a regression test, then, is given by the following formula.

$$\tau_{i,j}^{(k)} = \sum_{a=1}^{m} p_a^{(k)} \chi_a^{i,j}$$

where m represents the cardinality of $\{M_C \cup M_j\}$ as defined earlier. In this case, τ , is simply the expected value for code churn under the profile $P^{(k)}$.

This concept of test efficiency permits the numerical evaluation of a test on the actual changes that have been made to the software

system. It is simply the expected value of the fault exposure from one release to another under a particular test. If the value of τ is large for a given test then the test will have exercised the changed modules. If the set of τ 's for a given release is low then it is reasonable to suppose that the changed modules have not been tested in proportion to the number of probable faults that were introduced during the maintenance changes.

For practical purposes, we need to know something about the upper bound on test efficiency. That is, if we were to execute the best possible test, what then would be the value of test efficiency. A ***best*** regression test is one that will spend the majority of its time in the modules that have changed the most from one build to the next. Let,

$$\mathrm{X}^{i,j} = \sum_{a=1}^{n} \chi_a^{i,j} .$$

This is the total code churn between the i and j builds. To exercise each module in proportion to the change that has occurred in the module during its current revision, we will compute this proportion as follows:

$$q_a = \chi_a / \mathrm{X} .$$

This computation will yield a new hypothetical profile called the ***best profile***. That is, if all modules were executed in proportion to the amount of change that they had received we would then theoretically have maximized our exposure to software faults that may have been introduced.

Finally, we seek to develop a measure that will relate well to the difference between the actual profile that is generated by a test and the best profile. To this end, consider the following term, $|p_i - q_i|$. This is the absolute value between the best profile and the actual profile. This value has a maximum value of 1 and a minimum of 0. The minimum value will be achieved when the module best and actual coverage are identical. A measure of the total coverage for a set of modules (task or program) is then,

$$Coverage = 10 - 5 * \sum_i |p_i - q_i|$$

This coverage value has a maximum value of 10 when the best and the actual profiles are identical and 0 when there is a complete mismatch of profiles.

11 Regression Test Results

The following discussion documents the results of the execution of 36 instrumented tasks on two sequential builds of a large embedded software system. The perspective of this discussion is strictly from the standpoint of regression testing. That is, certain program modules have changed across the two sequential builds. The degree of this change is measured by code churn. As has been clearly demonstrated on the Cassini spacecraft project, the greater the change in a program module, the greater the likelihood that faults will have been introduced into the code by the change. Each of the regression tests, then, should attempt to exercise these changed modules in proportion to the degree of change. If a changed module were to receive little or no activity during the test process, then we must assume that the latent faults in the module will be expressed when the software is placed into service.

All of the tasks in system were instrumented with our Clic 1.0 tool. This tool would permit us to count the frequency of execution of each module in each of the instrumented tasks and thus obtain the execution profiles for these tasks for each of the tests. The execution profiles show the distribution of activity in each module of the instrumented tasks. For each of the modules, the code churn measure was computed. The code churn values for each modules reflected the degree of change of the modules during the most recent sequence of builds. The cumulative churn values for all tasks are shown in the second column of Table 14.

A churn value of zero indicates that the module in question received no changes during the last build sequence. A large churn value (>30) indicates that the module in question received substantial changes.

For the subsequent analysis, two profile values for each test will be compared. The ***actual profile*** is the actual execution profile for each test. The best profile is the best hypothetical execution profile given that each module would be tested directly in proportion to its churn value. That is, a module whose churn value was zero would receive little or no activity during the regression test process.

Table 14. Test Summary by Task

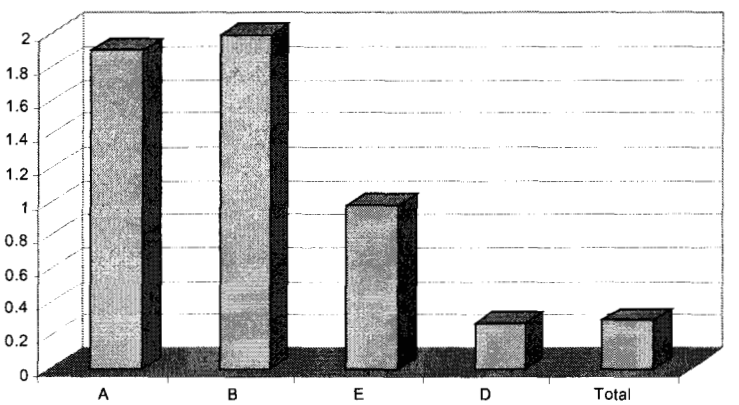
Task	Churn	Best Profile	Actual Profile	Best Coverage	Actual Coverage
A	2028.31	5.96E-01	1.94E-02	1208.266	39.291
B	487.26	1.43E-01	7.97E-03	69.73076	3.883045
C	154.72	4.54E-02	8.94E-04	7.031242	0.138295
D	150.77	4.43E-02	2.71E-01	6.676342	40.89712
E	150.71	4.43E-02	2.67E-03	6.671584	0.402485
F	126.46	3.71E-02	3.17E-03	4.697273	0.401035
G	121.00	3.55E-02	2.79E-03	4.299957	0.337566
H	117.20	3.44E-02	8.15E-04	4.034306	0.095532
I	14.38	4.23E-03	2.96E-04	0.060797	0.004252
J	9.11	2.68E-03	4.97E-05	0.024392	0.000453
K	6.84	2.01E-03	4.99E-01	0.013768	3.417803
L	6.27	1.84E-03	4.42E-05	0.011558	0.000277
M	5.64	1.66E-03	2.83E-03	0.00936	0.015955
N	5.17	1.52E-03	7.46E-05	0.00786	0.000386
O	3.92	1.15E-03	1.47E-04	0.004534	0.000579
P	3.90	1.15E-03	2.27E-03	0.004486	0.008868
Q	3.20	9.42E-04	1.72E-01	0.003018	0.552852
R	2.28	6.70E-04	2.12E-06	0.001528	4.83E-06
S	1.85	5.44E-04	7.07E-04	0.001006	0.001308
T	1.84	5.42E-04	6.40E-05	0.001001	0.000118
U	1.19	3.52E-04	4.48E-04	0.000422	0.000537
V	0.84	2.49E-04	8.63E-04	0.000212	0.000733
W	0.68	2.02E-04	1.17E-04	0.000138	8.04E-05
X	0.54	1.60E-04	3.81E-03	8.76E-05	0.00208
Y	0.26	7.82E-05	3.50E-03	2.08E-05	0.000931
Z	0.22	6.75E-05	1.86E-04	1.55E-05	4.27E-05
AA	0.09	2.83E-05	6.34E-05	2.73E-06	6.11E-06
AB	0.08	2.61E-05	1.55E-05	2.31E-06	1.38E-06
AC	0.04	1.30E-05	6.54E-07	5.78E-07	2.9E-08
AD	0	0.00E+00	1.09E-05	0	0
AE	0	0.00E+00	3.75E-06	0	0
AF	0	0.00E+00	3.71E-03	0	0
AG	0	0.00E+00	3.91E-07	0	0
AH	0	0.00E+00	2.15E-04	0	0
AI	0	0.00E+00	4.57E-07	0	0
AJ	0	0.00E+00	2.85E-06	0	0
	3404.946			1311.551	89.45334

From Table 14 we can seen that the A and B tasks have received the greatest change activity. Associated with each task entry in this table is the Best Profile and the Actual Profile for the task across all tests. The last row in the table gives the total values for code churn for all tasks. The last two columns of this table contain the expected value for the code churn of the task under the best profile and also under the actual profile. These columns are labeled Best Coverage and Actual Coverage. The total expected value for code churn under the best profile is 1311. The total expected value for code churn under the actual

profile is 89. The tests spent a disproportionate amount of time in modules that had not changed during this build interval. The ratio of Total Actual Coverage to Total Best Coverage will yield a percent coverage index for the task, for the system, or for the test depending on the granularity of the summary.

The change coverage index was computed by module for each task and then for the total system. In Figure 5, these coverage data are presented for the total system and Tasks A, B, D, and E. For this figure, the values have been scaled onto the interval from 0 to 10. Had there been perfect best coverage, the total value would have been 10. The coverage values for the A and B tasks were the best out of all tasks. The E and D tasks, while having relatively high code churn values, did not fare so well. The test coverage of the D task was typical of the total system, shown as the rightmost entry in this figure.

Figure 5 – Change Coverage Index



We would now like to look within a task to see why the A and B tasks showed better coverage than other tasks. The difference between the best profile and the actual profile is shown in Figure 6. Here, if the line is negative, this means that the module in question was exercised well out of proportion to the possible faults that it contained. On the other hand if the line is positive, then the module in question was not

exercised in proportion to the faults that it might contain. A perfect line on this chart would be perfectly straight at zero on the profile axis.

A similar graph is shown in Figure 7 for Task B. Here we can see that almost all test activity was on three distinct program modules (the negative values). The code that was changed was not exercised by this test to any large extent.

Figure 6 - Difference Between Best and Actual Profile for Task A

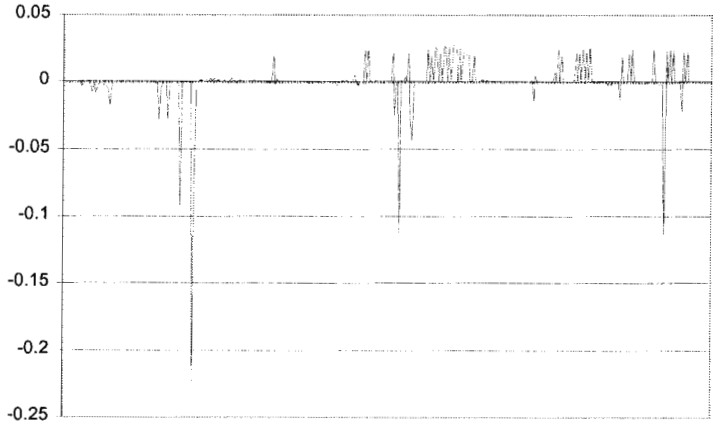
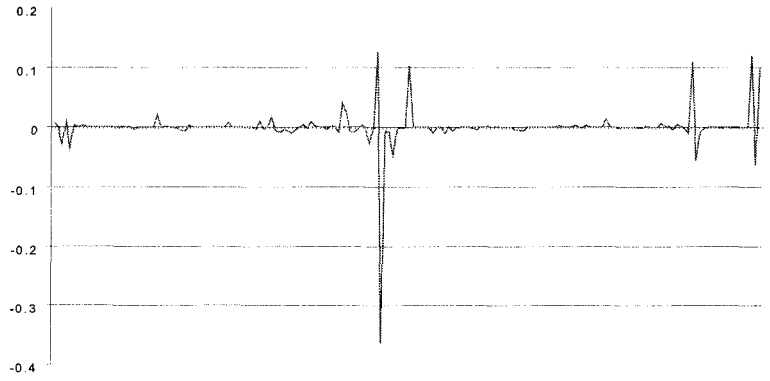


Table 15 summarizes the performance of the best 24 of suite of 115 instrumented tests. Only those tests whose performance index exceeded 10% of a theoretical total are shown here. Again the performance index shown in this figure was computed by forming the ratio of the actual profile to the best profile for that test. It must be remembered that not all tests will exercise all modules. The performance index is computed only for those modules whose functionality was included in the test. From a regression test perspective, we now know that we have a testing problem. None of these tests do a really good job in executing the code most likely to contain the newly introduced faults.

Figure 7 - Difference Between Best and Actual Profile for Task B



12 Summary

There is a distinct and a strong relationship between software faults and measurable software attributes. This is in itself not a new result or observation. The most interesting result of this current endeavor is that we also found a strong association between the fault insertion process over the evolutionary history of a software system and the degree of change that is taking place in each of the program modules. We also found that the direction of the change had an effect on the number of faults inserted. Some changes will have the potential of introducing very few faults while others may have a serious impact on the number of latent faults. Different numbers of faults may be inserted, depending upon whether code is being added to or removed from the system.

In order for the measurement process to be meaningful, the fault data must be very carefully collected. In this study, the data were extracted ex post facto as a very labor intensive effort. Since fault data cannot be collected with the same degree of automation as much of the data on software metrics being gathered by development organizations, material changes in the software development and software maintenance processes must be made to capture these fault data. Among other

things, a well defined fault standard and fault taxonomy must be developed and maintained as part of the software development process. Further, all designers and coders should be thoroughly trained in its use. A viable standard is one that may be used to classify any fault unambiguously. A viable fault recording process is one in which any one person will classify a fault exactly the same as any other person.

Table 15. Individual Test Summaries

Test #	Percent Coverage	Test #	Percent Coverage
28	20.6	177	11.7
18	19.0	31	11.6
14	18.2	3	11.5
12	16.9	167	11.5
47	14.8	59a	11.4
49	14.8	2	11.3
169	14.7	159	11.3
156	13.2	1	10.9
20	13.1	38	10.8
39	12.9	180	10.7
9	12.2	33	10.6
158	12.2	137	10.2

Finally, the whole notion of measuring the fault insertion process is its ultimate value as a measure of software process. The software engineering literature is replete with examples of how software process improvement can be achieved through the use of some new software development technique. What is almost absent from the same literature is a controlled study to validate the fact that the new process is meaningful. The techniques developed in this study can be implemented in a development organization to provide a consistent method of measuring fault content and structural evolution across multiple projects over time. The initial estimates of fault insertion rates can serve as a baseline against which future projects can be compared to determine whether progress is being made in reducing the fault insertion rate, and to identify those development techniques that seem to provide the greatest reduction.

Software test is not an intuitive process. Different modules are changed between builds. A regression test that was satisfactory for one build might well be totally inadequate on a subsequent build. When a program is subjected to numerous test suites to exercise differing aspects of its functionality, the test risk of a system will vary greatly as a result of the execution of these different test suites. Intuitively – and empirically – a program that spends a high proportion of its time executing a module set of high relative complexity will be more failure prone than one driven to executing program modules with low complexity values. Thus, we need to identify the characteristics of test scenarios that cause our criterion measures of χ and τ to be large.

The importance of this research is that we can now have a clearer understanding of how to quantify and evaluate the effectiveness of the regression testing process. For this study, we were not able to perform an analysis of test effectiveness on the same system for which we estimated the rate of fault insertion. We are currently working with NASA and commercial software development efforts to apply both types of analysis to the same project, with the goal of improving our ability to estimate the number of faults remaining in the system after the completion of a test sequence and allocate them among those portions of the system that have changed since the last increment.

Acknowledgements

The research described in this paper was carried out at the University of Idaho and the Jet Propulsion Laboratory, California Institute of Technology. The work at the University of Idaho was partially supported by a grant from the National Science Foundation. Portions of the work performed at JPL were sponsored by the U. S. Air Force Operational Test and Evaluation Center (AFOTEC) and the National Aeronautics and Space Administration's IV&V Facility.

Bibliography

[1] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M.-Y. Wong, "Orthogonal Defect Classification - A

Concept for In-Process Measurement," IEEE Transactions on Software Engineering, November, 1992, pp. 943-946.

[2] M. H. Halstead, *Elements of Software Science*. Elsevier, New York, 1977.

[3] "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std 729-1983, Institute of Electrical and Electronics Engineers, 1983.

[4] "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE Std 982.1-1988, Institute of Electrical and Electronics Engineers, 1989.

[5] "IEEE Standard Classification for Software Anomalies", IEEE Std 1044-1993, Institute of Electrical and Electronics Engineers, 1994

[6] T. M. Khoshgoftaar and J. C. Munson , "Predicting Software Development Errors Using Complexity Metrics," *IEEE Journal on Selected Areas in Communications* 8, 1990, pp. 253-261.

[7] T. M. Khoshgoftaar and J. C. Munson "A Measure of Software System Complexity and Its Relationship to Faults," In *Proceedings of the 1992 International Simulation Technology Conference*, The Society for Computer Simulation, San Diego, CA, 1992, pp. 267-272.

[8] S. G. MacDonell, M. J. Shepperd, P. J. Sallis, "Metrics for Database Systems: An Empirical Study," *Proceedings of the Fourth International Software Metrics Symposium*, November 5-7, 1997, Albuquerque, NM, pp. 99-107

[9] J. C. Munson and T. M. Khoshgoftaar "Regression Modeling of Software Quality: An Empirical Investigation," *Journal of Information and Software Technology*, 32, 1990, pp. 105-114.

[10] J. C. Munson and T. M. Khoshgoftaar "The Relative Software Complexity Metric: A Validation Study," In *Proceedings of the Software Engineering 1990 Conference*, Cambridge University Press, Cambridge, UK, 1990, pp. 89-102.

[11] J. C. Munson and T. M. Khoshgoftaar "The Detection of Fault-Prone Programs," *IEEE Transactions on Software Engineering*, SE-18, No. 5, 1992, pp. 423-433.

[12] J. C. Munson, "Software Measurement: Problems and Practice," *Annals of Software Engineering,* J. C. Baltzer AG, Amsterdam 1995.

[13] J. C. Munson, "Software Faults, Software Failures, and Software Reliability Modeling," *Information and Software Technology*, December, 1996.

[14] J. C. Munson and D. S. Werries, "Measuring Software Evolution," *Proceedings of the 1996 IEEE International Software Metrics Symposium* , IEEE Computer Society Press, pp. 41-51.

[15] J. C. Munson and G. A. Hall, "Estimating Test Effectiveness with Dynamic Complexity Measurement," *Empirical Software Engineering Journal.* Feb. 1997.

[16] J. Neter, W. Wasserman, M. H. Kutner, Applied Linear Regression Models, Irwin: Homewood, IL, 1983

[17] A. P. Nikora, N. F. Schneidewind, J. C. Munson, "IV&V Issues in Achieving High Reliability and Safety in Critical Control System Software," proceedings of the International Society of Science and Applied Technology conference, March 10-12, 1997, Anaheim, CA, pp 25-30.

[18] A. P. Nikora, "Software System Defect Content Prediction From Development Process And Product Characteristics," Doctoral Dissertation, Department of Computer Science, University of Southern California, May, 1998.

[19] "User's Guide for UX-Metric 4.0 for Ada," SET Laboratories, Molino, OR, © SET Laboratories, 1987-1993.